Self-Stabilizing Spanning Tree Algorithm With a New Design Methodology

Vijay K. Garg *
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
garg@ece.utexas.edu

Anurag Agarwal
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-0233, USA
anurag@cs.utexas.edu

Abstract

Maintaining spanning trees in a distributed fashion is central to many networking applications. In this paper, we propose a self-stabilizing algorithm for maintaining a spanning tree in a distributed fashion for a completely connected topology. Our algorithm requires a node to process $O(1)$ messages on average in one cycle as compared to previous algorithms which need to process messages from every neighbor, resulting in $O(n)$ work in a completely connected topology. Our algorithm also stabilizes faster than the previous approaches. Our approach demonstrates a new methodology which uses the idea of *core* and *non-core* states for developing self-stabilizing algorithms. The algorithm is also useful in security related applications due to its unique design.

Keywords: Fault Tolerance, Self-Stabilization, Spanning Tree

1 Introduction

Fault tolerance is a major concern in distributed systems. The self-stabilization paradigm, introduced by Dijkstra [Dij74], is an elegant and a powerful mechanism for fault tolerance. Self-stabilizing systems tolerate transient *data faults* that can corrupt the state of the system. They ensure that a system starting from any state would converge to a legal state provided the faults cease to occur.

Self-stabilizing algorithms for spanning tree construction have been extensively studied. Spanning trees have many uses in computer networks. Once a spanning tree is established in a network, it may be used in broadcast of a message, convergecast, β synchronizer, and many other algorithms. As a result, it is desirable to have an efficient self-stabilizing algorithm for spanning trees. The first algorithm in this area was given in [DIM89, DIM90] which deals with building BFS tree for a graph. Other algorithms were also proposed for self-stabilizing BFS trees which dealt with different system models and assumptions [AKY91],[AG94], [HC92]. Algorithms have also been proposed for other types of trees — such as DFS tree [CD94] and minimum spanning tree [AS97]. There have been other papers which try to optimize on the memory used and stabilization time [Joh97], [AK93].

In this paper, we demonstrate a new technique for constructing self-stabilizing algorithms by using it for maintaining spanning trees. It is well known that one can design self-stabilizing algorithms with *detection* and *reset* strategy [AG94]. In this strategy, the nodes periodically test if the system is in legal state and on detection of a fault, carry out the reset strategy. Many self-stabilizing algorithms have *local detection*, i.e., the detection by each node corresponds to evaluation of a boolean predicate only on its and its neighbors' variables. The reset procedure may be complicated depending upon the application.

Our method is an extension of the above strategy. We view the set of *global* states as cross-product of *core* states and *non-core* states. The core states satisfy the property: There exists a legal state for every *core* state. The *non-core* component of a global state is maintained only for performance reason. Given the *core* component, one could always recreate the *non-core* component. In our algorithm for maintaining a spanning tree, we will

*supported in part by the NSF Grants ECS-9907213, CCR-9988225, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant.

use Neville's code [Nev53] of the tree as the *core* component and *parent* structure as the *non-core* component. Given any Neville's code, there exists a unique labeled spanning tree in a completely connected graph. Now assume that our program suffers from a data fault. The data fault could be in the core component or the non-core component. However, every bit pattern in the core component results in a valid code. Therefore, in either case, we will assume that it is the non-core component that has changed. Upon detecting that the non-core component does not correspond to the core component, we simply reset the non-core component to a value corresponding to the core component. The challenge lies in efficient detection and reset of the state when information is distributed across the network. Similarly, for maintaining a link list of nodes in a network, the *inversion vector* of the permutation corresponding to the linked list can be the core component and then some auxiliary data structures can be maintained to make the fault detection and correction efficient.

We assume our system to be a completely connected graph with n nodes having ids $1 \dots n$. Our algorithm is designed for asynchronous message-passing systems, and unlike many other self-stabilizing algorithms, it does not require a central daemon [Dij74] for scheduling decisions. Although some of our assumptions are stronger than the previous work, our algorithm has some significant advantages. All the self-stabilizing algorithms invariably consist of a "loop" that is executed infinitely often. We would refer to the steps performed in one iteration of the loop to be a *cycle*. Previous algorithms require examining every neighbor's variables during correction, needing $O(n)$ steps in one cycle for a completely connected topology. On the other hand, in our algorithm each process executes $O(1)$ steps on average in one cycle.

Traditionally, the notion of time used in self-stabilizing asynchronous algorithms is the number of *asynchronous rounds* [DIM91] the algorithm executes. Informally, an asynchronous round is a sequence of steps executed by the processes during the execution of the algorithm such that every process takes at least one step in that sequence. Our algorithm does not fit in well with model of asynchronous rounds due to presence of asynchronous receives. We use a model similar to synchronous model for evaluating the time complexity of our algorithm. In this model, the stabilization time of the previous algorithms remains the same as that in asynchronous rounds model. In the new model, the stabilization time of our algorithm is $O(d)$, where d is an upper bound on the number of times a node appears in the Neville's code. It turns out that d is $O((\log n)/\log\log n)$ with very high probability for a randomly chosen code. This gives a very small stabilization time.

As a result of using the idea of *core* and *non-core* states, we also provide the individual nodes with the ability to systematically change the structure of the tree. This renders the algorithm useful in settings not traditionally associated with self-stabilizing systems. As an example, we consider an application where the participating nodes would like to periodically change the structure of the tree in a distributed fashion for security purposes. This can be done using our algorithm by changing code value for some node which would result in a different tree upon stabilization. Another useful feature of our algorithm is that it allows the root of the tree to change dynamically. This is different from most of the previous approaches where the root node executes a different algorithm from the rest of the nodes, resulting in a fixed root. We also discuss an application which requires this feature. In summary, the main advantages of our algorithm are:

- Fast stabilization.
- Allows systematic change in tree structure.
- Root node can change dynamically.

The paper is organized as follows. Section 2 describes our system model in detail. Section 3 introduces the Neville's code and the algorithms for mapping Neville's code to a spanning tree and vice versa. In Section 4, we explain the data structures and the constraints on the data structures that are used to maintain the spanning tree. Section 5 explains the approach used for maintaining each of the constraints and our algorithm for maintaining the spanning tree. Section 6 presents some applications which could use our algorithm in a fault-free scenario.

2 System Model

We assume that the network is a completely connected graph with *n processes* having ids from 1 to n and this graph does not change during the operation of the algorithm. The processes in the system would be referred to as $P_1 \dots P_n$. Our algorithm is designed for reliable asynchronous message-passing systems. Each *process* maintains some *local variables*. The processes are connected to each other through point to point channels and communicate by passing messages to each other. The *configuration* c of the system is described by the values of the *local variables* for the processes and the *messages* present in the channels. A *computation step* consists of internal computation and a single communication operation: a send or receive. Now on, we would use the term *step* to refer to a computation step. A step a is said to be applicable to a configuration c iff there exists a configuration c' such that c' can be reached from c by a single step a. An execution $E = (c_1, a_1, c_2, a_2, \ldots)$ is an alternating sequence of configurations and steps such that c_i is obtained from c_{i-1} by the execution of the step a_{i-1}. A *fair* execution is an execution in which every step that is applicable infinitely often is executed infinitely often.

For measuring the time complexity of our algorithm, we use a model similar to the synchronous message passing system. In every unit of time, each process would execute exactly one step. Moreover, we assume that a message is available to the destination for receive in the step next to the one in which the message was sent and these messages are delivered to the destination in the same order in which they are received. The stabilization time of the algorithm is then given in terms of the number of time units required by the algorithm to stabilize. The reason for choosing such a model is explained later in the section.

Our algorithm has an *asynchronous receive* which requires special treatment. An asynchronous receive is a receive for which a process does not wait but on arrival of a relevant message, the process is notified through some interrupt. Asynchronous receives are handled by the system in the following way: A process executes the sequential steps (steps other than the asynchronous receive) of the algorithm whenever there are no outstanding messages in its channels. In case there are messages for asynchronous receives to be handled, the process alternates between the sequential algorithm and the message receives. We assume that the underlying provides us with the functionality to selectively receive messages based on the type and sender of the message. This requirement stems from our need to distinguish between messages intended for asynchronous receives and other messages.

We have chosen a different model for evaluation of our strategy as it allows a fair modeling of the asynchronous receives. In the previous works, the *asynchronous rounds* [DIM91] model was used. The first asynchronous round (or round) in an execution E is the shortest prefix E' of E such that each processor executes at least one step in E'. Let E'' be the suffix of E that follows E', $E = E'E''$. The second round of E is the first round of E'', and so on. The stabilization time of an algorithm is the maximum number of rounds it executes before the system reaches a legal state. In this model, a process waiting for a message would receive the message in one round whereas if the message receive is asynchronous, it fails to provide any guarantees. In practice, running time of both the algorithms depends upon the message delivery time in a similar way and hence their time complexities should be comparable. We try to achieve this by putting a bound on the message delivery time. Moreover, all the self-stabilizing algorithms are structured as a loop that is executed periodically. We would refer to this loop as a *cycle*. In practice, most of the applications would have a sufficiently large gap between two cycles to allow all the processes to complete one cycle and hence the cycles can be considered as synchronized with each other. Computing the time complexity in our model would give a better idea of the actual running time of the algorithm.

3 Neville's Third Encoding

To maintain a spanning tree (in a non-stabilizing manner), it is sufficient for each process to maintain the *parent* variable but this method is not self-stabilizing as a fault in the parent pointer of some process may result in an invalid structure.

For the purpose of this paper, all spanning trees rooted at P_n constitute the set of legal structures. We represent a tree through an encoding for labeled trees called the *Neville's third encoding* [Nev53] (The reader can find a discussion on other labeled tree encodings and their properties in [DM01]). In this paper the term "Neville's code" refers to Neville's third code. Each labeled spanning tree has a one-to-one correspondence with a Neville's code. This code is a sequence of $n-2$ numbers from the set $\{1 \ldots n\}$. Let Neville's code of the tree be denoted by $code[i]$ for $i \in \{1 \ldots n-2\}$. For completeness sake, derivation of Neville's code from a labeled spanning tree is discussed. Given a labeled spanning tree with n nodes, the Neville's code can be obtained by deleting $n-1$ edges in the tree as shown in Figure 1.

The algorithm starts by deleting the least leaf node (leaf with least label). At iteration i, a node $x[i]$ is deleted and its neighbor $y[i]$ is recorded. The edge between $x[i]$

```
x[1] = least node with degree 1;
for (i = 1; i < n; i++)
        y[i] = neighbor of x[i];
        delete edge between x[i] and y[i];
        if (degree[y[i]] == 1)
              x[i+1] = y[i];
        else
              x[i+1] = least node with degree 1;
```

Figure 1: Algorithm to compute Neville's code of a labeled tree

and $y[i]$ is deleted as well. Then, variable $x[i+1]$ is set to $y[i]$ if the degree of $y[i]$ is 1, otherwise it is set to the least leaf node. The sequence $\{y[i] | 1 \leq i \leq n-2\}$, thus obtained, is called Neville's code. Note that even though there are $n-1$ iterations, we only consider $n-2$ entries. It can be easily verified that in the iteration $n-1$, the value of $y[i]$ is always n. Although, the actual Neville's code has only $n-2$ entries, for all the algorithms in this paper we consider the $n-1$ length code which includes n at the end.

As an example, consider the labeled tree given in Figure 2. To compute the Neville's code for the tree, we start by deleting the least leaf node, 1. Since the parent of 1 is 5, at this point the code is (5). Now 5 is still not a leaf, so we again choose the least leaf node in the remaining tree, 3. We proceed by deleting 3 and adding its parent 2 to the code. Now 2 has become the leaf node in the tree. So 2 would be deleted and 7 would be added to the code. At this point code is $(5,2,7)$. Since 7 is not a leaf node, so the least leaf node, 4, would be deleted next and its parent 5 would be added to the code. Similarly, in the next iteration, 6 would be deleted and 5 would be added to the code. Now, 5 becomes a leaf node and it would be deleted. Its parent 7 would be added to the code. Since we have completed $n-1=6$ iterations of the algorithm, we stop at this point and the code $(5,2,7,5,5,7)$ is obtained.

Given Neville's code, the labeled spanning tree can also be computed easily. We first calculate the degree of each node v in the labeled spanning tree as follows:
degree(v) = 1 + number of times v appears in the code.
Note that for the root node n, this gives a value which is one higher than the actual degree of the root but this is required for the correctness of the algorithm. Once degree

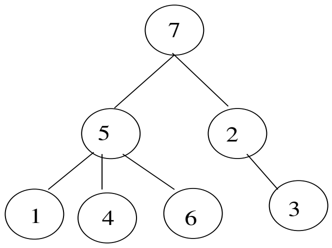

Figure 2: A spanning tree with Neville's code (5,2,7,5,5,7)

of each node is known, the procedure given in Figure 3 can be used to compute the code. In the given procedure $n-1$ edges are constructed.

```
j = least node with degree 1;
for (i = 1; i < n; i++)
        parent[j] = code[i];
        degree[j]--;
        degree[code[i]]--;
        if (degree[code[i]] == 1) then
              j = code[i];
        else
              j = least degree node with degree 1;
```

Figure 3: Algorithm to compute labeled tree from Neville's code

We require P_i to maintain $code[i]$ as the core data structure. If efficiency were not an issue, this would be sufficient for a self-stabilizing algorithm. Periodically, all nodes would send their *code* to P_n. P_n would calculate $parent[i]$ for each node P_i and send it back. P_i would reset $parent[i]$ to the value received from P_n. Even if $parent[i]$ was corrupted, it would be reset to agree with the spanning tree given by Neville's code. If the variable $code[i]$ gets changed, it would still result in a valid spanning tree. The *parent* pointers would then be reset to agree with the new code.

This method would be wasteful when there are no errors. The transmission of code to the root node and the transmission of the parent pointer to various nodes will not change anything. In a large network, it is desirable to have local detection of error and only on an error, the correction algorithm should be invoked.

4 Non-Core Data Structures for Spanning Trees

Our strategy would be to introduce new data structures in the system so that by imposing a set of constraints on these data structures, we can efficiently detect and correct data faults. For this purpose, the following data structures are added:

- *parent*: The variable $parent[i]$ gives the parent of node P_i in the spanning tree.
- f: The variable $f[i]$ gives us the iteration in which the node P_i is deleted in the algorithm for obtaining Neville's code of a tree. Therefore, $code[f[i]]$ gives us the $parent[i]$. Since P_n is not deleted in first $n-1$ iterations, we assume that $f[n] = n$.
- z: The variable $z[i]$ gives the largest value of j such that $code[j] = i$. If there is no such j, then $z[i] = 0$.

Based on the properties of Neville's code, it can be verified that the variables — *code*, *parent*, f and z — satisfy the following constraints:

(R1) $\forall i : code[f[i]] = parent[i]$
Follows from the property of the f relating it to the parent.

(R2) $(\forall i : 1 \leq i \leq n-2 \Rightarrow 1 \leq code[i] \leq n)$
$\wedge (code[n-1] = n) \wedge (code[n] = 0)$
This constraint is the definition of code extended to all the nodes.

(R3) (i) $\forall i : 1 \leq i < n \Rightarrow 1 \leq f[i] \leq n-1$
This constraint puts restriction on the range of values that a node other than the root is allowed to take.

(ii) f is a permutation on $[1 \ldots n]$
The definition of f along with the topology of the algorithm in Figure 1 imply that the f values are distinct and range from $1 \ldots n$.

(R4) $\forall j : z[j] = \max\{i | code[i] = j\} \cup \{0\}$
This is the definition of z.

(R5) $\forall i : z[i] \neq 0 \Rightarrow (f[i] = z[i] + 1)$
If a node i, which at the starting of the algorithm was not a leaf node, becomes a leaf node during the iteration j of the algorithm, then it is deleted in the iteration $j+1$. This constraint enforces this condition.

(R6) $\forall i, j : (z[i] = 0) \wedge (z[j] = 0) \wedge (i < j) \Rightarrow (f[i] < f[j])$
This constraint says that the nodes with $z = 0$ should finish in increasing order of their labels.

These constraints are strong enough to characterize a spanning tree, i.e., given a set of data structures *code*, *parent*, f and z which satisfy these constraints, the *parent* structure results in a valid spanning tree regardless of the definitions of these data structures. From now on, when we consider the data structures *code*, *parent*, f and z, we would just think of them as obeying a certain set of constraints and not necessarily corresponding to the original definitions that were given for them.

We would be dealing with three sets of constraints — $\mathcal{R} = \{R1, R2, R3(i), R4, R5\}$ and $\mathcal{C} = \{R1, R2, R3, R4, R5\}$ and $\mathcal{E} = \{R1, R2, R3, R4, R5, R6\}$. It is evident that any algorithm which satisfies the constraint set $\mathcal{C}$ would also satisfy the constraint set $\mathcal{R}$ and similarly, any algorithm which satisfies the constraint set $\mathcal{C}$ would also satisfy the constraint set $\mathcal{E}$. The trees resulting from obeying these constraint sets possess different guarantees. The theorems that follow provide a characterization of those guarantees.

Theorem 1 *If code, parent, f and z satisfy constraint set $\mathcal{R}$ then parent forms a valid spanning tree rooted at P_n.*

Proof: Let the directed graph formed by the *parent* relation satisfying constraints $\mathcal{R}$ be T_{parent}. The edges of T_{parent} are directed from the child to the parent.
We first show that T_{parent} is acyclic. Let $i = parent[j]$ in T_{parent} for some nodes i and j. Then,

$code[f[j]] = i$	(Using (R1))
$\Rightarrow (z[i] \neq 0) \wedge (f[j] \leq z[i])$	(Using (R4))
$\Rightarrow f[j] < f[i]$	(Using (R5) for j)

Applying this argument repeatedly shows that ancestor of a node has higher f value than the f value for the node itself. This implies that no node is ancestor of itself and hence T_{parent} is acyclic.

We now show that every node except P_n has outdegree 1 and P_n has outdegree 0. Consider a node $i \neq n$. Then,

$f[i] \neq n$	(Using (R3)(i))
$\Rightarrow 1 \leq code[f[i]] \leq n$	(Using (R2))
$\Rightarrow 1 \leq parent[i] \leq n$	(Using (R1))

This implies that in T_{parent}, every node except P_n has outdegree 1. For P_n, consider the following:

$$
\begin{array}{ll}
code[n-1] = n \wedge code[n] = 0 & \text{(Using (R2))} \\
\Rightarrow z[n] = n-1 & \text{(Using (R4))} \\
\Rightarrow f[n] = z[n]+1 & \text{(Using (R5))} \\
\Rightarrow f[n] = n & \\
\Rightarrow code[f[n]] = 0 & \text{(Using (R2))} \\
\Rightarrow parent[n] = 0 & \text{(Using (R1))}
\end{array}
$$

Therefore, P_n does not have a parent. Since all other nodes have a parent within the range $1 \dots n$ and there are no cycles in T_{parent}, T_{parent} forms a spanning tree rooted at P_n.

■

The above theorem just ensures that the *parent* forms a spanning tree. It does not enforce any relationship between the structure of the tree formed by *parent* and tree corresponding to *code*. The next theorem establishes this relationship.

Theorem 2 *If code, parent, f and z satisfy constraint set $\mathcal{C}$, then parent forms a rooted spanning tree isomorphic to the tree generated by code.*

Proof: Since *code*, *parent*, f and z satisfy the constraint set $\mathcal{C}$, they also satisfy the constraint set $\mathcal{R}$. Hence, by Theorem 1, *parent* forms a spanning tree rooted at P_n.

Consider the following two trees:

(1) T_{parent}: The tree formed by *parent* which satisfies $\mathcal{C}$ and

(2) T_{code}: The tree generated using *code* by the algorithm given in Figure 3.

We define data structures $parent'$, f' and z' for T_{code}. Variable $parent'[i]$ represents the parent of node i in T_{code}, $f'[i]$ gives the iteration in which the node i is assigned its parent during the execution of algorithm for building T_{code} from *code* and $z'[i]$ gives the last occurrence of i in *code*. Note that since the constraint (R4) is the same as definition for z', $\forall i : z[i] = z'[i]$. We would be using z instead of z' for the rest of the proof. Both f and f' are permutations on $1 \dots n$. This implies that $\forall i (\exists j : f[i] = f'[j])$ and moreover, this j is unique. This allows us to define an isomorphism function, $M : [n] \rightarrow [n]$ as:

$$M(i) = j \text{ such that } f[i] = f'[j]$$

Now, T_{parent} and T_{code} are isomorphic *iff*

$$\forall i, j : (i = parent[j]) \Rightarrow M(i) = parent'[M(j)]$$

We prove the above condition by showing that $\forall i, j : (i = parent[j]) \Rightarrow M(i) = i$ followed by proving that $\forall i, j : (i = parent[j]) \Rightarrow i = parent'[M(j)]$. The data structures $parent'$, f' and z obey the constraint set $\mathcal{C}$. Consider a node $i = parent[j]$ for some nodes i and j. Then,

$$
\begin{array}{ll}
i = parent[j] & \\
\Rightarrow z[i] \neq 0 & \text{(Using (R1) and (R4))} \\
\Rightarrow f'[i] = z[i]+1 & \text{(Using (R5) for } f'\text{)} \\
\Rightarrow f[i] = f'[i] & \text{(Using (R5) for } f\text{)} \\
\Rightarrow M(i) = i & \text{(Definition of } M\text{) — (1)}
\end{array}
$$

Node i also satisfies the following property:

$$
\begin{array}{ll}
i = code[f[j]] & \text{(Using (R1) for } f\text{)} \\
\Rightarrow i = code[f'[M(j)]] & \text{(Definition of } M\text{)} \\
\Rightarrow i = parent'[M(j)] & \text{(Using (R1) for } f'\text{) — (2)}
\end{array}
$$

Conditions (1) and (2) together prove the required isomorphism condition and hence the two trees T_{parent} and T_{code} are isomorphic.

■

In the next theorem, we show that by maintaining the constraint set $\mathcal{E}$, it is possible to maintain a tree which is exactly same as the spanning tree generated by *code*.

Theorem 3 *If code, parent, f and z satisfy constraints set $\mathcal{E}$, then the rooted spanning tree formed by parent is same as the tree generated by code.*

Proof: Since $\mathcal{C} \prec \mathcal{E}$, therefore the two trees T_{parent} and T_{code} as defined in Theorem 2 are isomorphic. In fact, we showed an even stronger property - The internal nodes of the tree T_{parent} are mapped to themselves in tree T_{code}. So for proving that the trees T_{parent} and T_{code} are the same, we just need to show that the leaf nodes in T_{parent} map to themselves in T_{code} or in other words $\forall i : z[i] = 0 \Rightarrow M(i) = i$. Let $\mathcal{L} = [n]/\{i : z[i] \neq 0\}$. Since M is a bijection and $\forall i : z[i] \neq 0 \Rightarrow M(i) = i$, so M is a bijection on $\mathcal{L}$ as well. When constructing the Neville's code for a tree, we always choose the *least* leaf node whenever we need to pick a leaf node. This ensures that $\forall i, j : (z[i] = 0) \wedge (z[j] = 0) \wedge (i < j) \Rightarrow (f'[i] < f'[j])$, which is the condition equivalent to (R6) with respect to f' instead of f. This implies that the order of finishing for the leaves in the two trees is the same which combined with the condition of M being a bijection on $\mathcal{L}$, proves the required condition.

i	1	2	3	4	5	6	7
parent	2	7	5	5	7	5	0
code	5	2	7	5	5	7	0
f	2	3	1	4	6	5	7
z	0	2	0	0	5	0	6

Table 1: Example of structures *parent*, *code*, f and z satisfying the constraints (R1)-(R5)

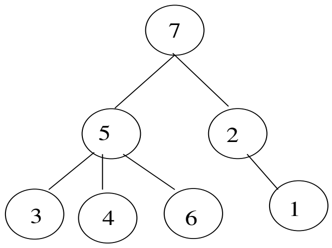

Figure 4: Tree corresponding to *parent* given in table 1

■

The above theorem suggests that there is a possibility that the tree formed by *parent* is not the same as the tree generated by the *code*. For example, consider the value of the variables given in Table 1. It can be easily verified that these values satisfy the constraint set $\mathcal{C}$. The tree corresponding to the *code* is the one we considered earlier in Figure 2. The tree generated by *parent* is shown in Figure 4. The two trees are not the same but they are isomorphic.

5 Maintaining Constraints

Each node i maintains $parent[i]$, $code[i]$, $f[i]$ and $z[i]$ and cooperate to ensure that the required constraints are satisfied, resulting in a valid rooted spanning tree. We present a strategy for efficient detection and correction of the faults for each of the constraints (R1)-(R5). We will consider the constraint (R6) later in the paper.

5.1 Constraints (R1) and (R2)

The constraint (R1) is trivial to check locally. Each node i inquires node $j = f[i]$ for $code[j]$. If this value does not match $parent[i]$, then the constraint $(R1)$ is violated. On violation, (R1) can be ensured by setting $parent[i]$ to $code[j]$. The constraint (R2) is also trivial to check and correct locally.

5.2 Constraint (R3)

Constraint (R3)(i) is a local constraint which can be checked easily. Violation of this constraint can be fixed by simply setting f to a random number between 1 and $n-1$. Constraint (R3)(ii) requires f to be a permutation on $1 \ldots n$. This can, in turn, be modeled in terms of the following constraints:

(C1) $\forall i : 1 \leq f[i] \leq n$

(C2) $\forall i, j : f[i] \neq f[j]$

The violation of (C1) is easy to detect. Every node i checks the value $f[i]$ periodically. If it is not between 1 and n, then a fault has occurred. The constraint (C2) is more interesting. At first glance it seems counter-intuitive that we can detect violation of (C2) in $O(1)$ messages. However, by adding auxiliary variables, the above task can indeed be accomplished. We maintain $g[i]$ at each process P_i such that, in a legal global state $f[i] = j \equiv g[j] = i$. Thus, g represents the inverse of the array f. Note that the inverse of a function exists iff it is one-one and onto which is true in this case. If each process P_i maintains $f[i]$ and $g[i]$, then it is sufficient for a node to check periodically the following constraints:

(D1) $\forall i : 1 \leq f[i] \leq n$

(D2) $\forall i : 1 \leq g[i] \leq n$

(D3) $g[f[i]] = i$

It is easy to show that (C2) is implied by (D1)-(D3). If for some distinct i and j, $f[i]$ is equal to $f[j]$, then $g[f[i]]$ and $g[f[j]]$ are also equal. This means that $(g[f[i]] = i)$ and $(g[f[j]] = j)$ cannot be true simultaneously. (D3) can be checked by P_i by sending a message to $P_{f[i]}$ periodically, prompting $P_{f[i]}$ to check whether $g[f[i]] = i$ is true. Note that by introducing additional variables we have also introduced additional sources of data faults. It may happen that requirements (C1)-(C2) are met, but due to faults in g, constraints (D1)-(D3) are not met. We believe that the advantage of local detection of a fault outweighs this disadvantage.

The above scheme has an additional attractive property: If we assume that there is a single fault in f or g, then it can also be automatically corrected as shown next.

The function g being inverse of f also implies that f is inverse of g. This implies that the following constraint (D4) is also met for a fault-free data structure:

(D4) $f[g[i]] = i$

```
Pi::
  var
      f, g: array[1..n] of integer;

  Periodically do
      if (g[f[i]] ≠ i) ∧ f[g[f[i]]] ≠ f[i]
            g[f[i]] = i
      if (f[g[i]] ≠ i) ∧ g[f[g[i]]] ≠ g[i]
            f[g[i]] = i
```

Figure 5: Implementation of Permutation with local correction of 1 fault

Now assume that a node i discovers that $g[f[i]] \neq i$. This means that either $f[i]$ or $g[f[i]]$ got corrupted. To detect which of the case has happened, it is sufficient to check whether

$$f[g[f[i]]] = f[i]$$

If the above equation does not hold, then $g[f[i]]$ is corrupted and is set back to i. If the above equation holds, but $g[f[i]] \neq i$ then $f[i]$ is corrupted and it needs to be reset. What value should $f[i]$ be set to? We need to set it to k such that $g[k]$ equals i. This correction would be done by node k because node k will find that $f[g[k]] \neq k$. Hence, by the similar reasoning as above node k will deduce that $f[g[k]]$ is corrupted and will reset it to k. The program for P_i is shown in Figure 5. For simplicity, we let process P_i simply read and write variables of other processes. In practice, this may be translated into messages. Note that in our scheme a permutation may undetectably change into another permutation (when there are multiple faults) but if f is not a valid permutation, the violation will be detected.

5.3 Constraint (R4)

This constraint can be modeled in terms of the following constraints:

(E1) $\forall i : (z[i] \neq 0) \Rightarrow (code[z[i]] = i)$

(E2) $\forall i, j : (code[j] = i) \Rightarrow (z[i] \geq j)$

For checking (E1), node i prompts the node $z[i]$ to verify that $code[z[i]] = i$. If the check fails, then $z[i]$ can be set to 0, which may not be the correct value for $z[i]$. If $z[i]$ is set incorrectly to 0, then constraint (E2) would also be violated. As a result, while checking for (E2), $z[i]$ would be set appropriately. For checking (E2), every node j sends a message to node $code[j]$ to verify that $z[code[j]] \geq j$. If (E2) is found to be violated upon receiving a message from node j, then $z[code[j]]$ is set to j.

5.4 Constraint (R5)

The constraint (R5)

$$\forall i : z[i] \neq 0 \Rightarrow (f[i] = z[i] + 1)$$

can be checked locally and on detection of a fault, $f[i]$ can be set to $z[i] + 1$.

5.5 Complete Algorithm

Depending upon the set of constraints ($\mathcal{R}$ or $\mathcal{C}$) that a process obeys, we have two versions of the algorithm. They differ in the guarantees about the resulting tree and the time complexity of the algorithm.

5.5.1 Maintaining $\mathcal{R}$

As we proved in Theorem 1, the set of constraints $\mathcal{R}$ is sufficient to maintain a spanning tree. The complete algorithm for process i to maintain the constraint set $\mathcal{R}$ is given in the Figure 7. We will refer to this algorithm as *SSR*. In the algorithm, instead of denoting variables like $code[i]$, we have used $P_i.code$ to emphasize that the variables are local to the processes and are not shared. The algorithm checks the constraints one by one and on the violation of a constraint, it takes corrective action. For checking constraints which involve obtaining the value of another process's variable, we have used a primitive *get*. This involves the sender sending a request for the required variable and the receiver then replying with the appropriate value. So a *get* operation would involve two messages being exchanged. Most of the algorithm follows directly from the checks required for a constraint. The interesting thing to note is the presence of an asynchronous receive for the messages of type "Check z". These asynchronous receives are crucial for the efficiency of the algorithm and they require special attention during the analysis of the algorithm in the following section.

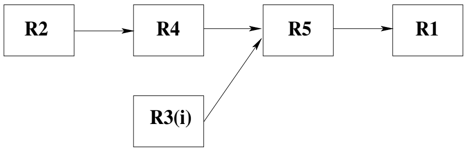

Figure 6: Stabilization order of the constraints

5.5.2 Analysis of *SSR*

In this section, we give the proof of correctness of the algorithm and its stabilization time in terms of the number of time units for stabilization. The following lemma is based on the structure of the algorithm *SSR* given in Figure 7.

Lemma 1 *Each process performs all the actions in one cycle in every $O(1)$ time units.*

Proof: In each iteration of the loop there are 2 "get" operations (one of which may not be executed by all the processes) and 1 "send" operation (which again may not be performed by all the processes), along with the asynchronous receive operations. As discussed earlier, each "get" operation consists of 2 communication steps. This results in a total of 5 communication steps. Since the asynchronous receives may alternate with these 5 steps, we are guaranteed that within 10 steps every process will complete one iteration of the main loop. Hence, the result follows.

■

For the purpose of the proof we define the following terms:

Definition: A variable v is said to be *stabilized* by an algorithm if the value of v does not change after some finite number of rounds of the algorithm irrespective of the initial state of the algorithm.

Definition: A constraint R is said to be *stabilized* by an algorithm if R never becomes false after some finite number of rounds of the algorithm irrespective of the initial state of the algorithm

The proof of correctness is given by the lemmas and the theorems that follow. The lemmas should be considered in the order given as a lemma assumes the results of the previous lemmas. The order of the lemmas corresponds to the order in which we stabilize the constraints. This order is also given in Figure 6

Lemma 2 *The algorithm SSR stabilizes "code" and (R2) for all the processes by $O(1)$ rounds.*

Proof: From the algorithm, it is clear that the *code* variable is only changed while checking the constraint (R2) and after one such change, *code* always satisfies the constraint (R2). So after one execution of the check for (R2), the *code* variable is never changed. From Lemma 1, each process executes all the actions in one cycle in $O(1)$ time units and thus the *code* variable would be set correctly in $O(1)$ time units. Hence, *SSR* stabilizes *code* and (R2) in $O(1)$ time units.

■

Lemma 3 *The algorithm SSR stabilizes z and (R4) in $O(d)$ time units, where d is the upper bound on the number of times a node appears in the code.*

Proof: Consider the execution of the algorithm once (R2) has stabilized. Constraint (R4) can be violated in two ways : (1) $\exists i : (z[i] = k) \wedge (code[k] \neq i)$ or (2) $\exists i : (z[i] = k) \wedge (code[k] = i) \wedge (\exists j > k : code[j] = i)$. Note that in this case, the check is dependent only upon the value of *code*. Consider the first check for (R4) given in the algorithm. If the violation is of the first kind, then P_i would be able to check that $code[z[i]] \neq i$ and would reset $z[i] = 0$. From Lemma 1, every process would be able to perform this step in $O(1)$ time units after stabilization of (R2), still requiring $O(1)$ time units in all. We say that at this point (R4) has *partially stabilized*.

Let $z_c[i]$ be the correct value of $z[i]$ according to constraint (R4). After partial stabilization of (R4), we are guaranteed that $\forall i : z[i] \leq z_c[i]$. Now consider the messages of the type "Check z" sent by the processes. A process P_j sends a message "Check z" to node P_i iff $code[j] = i$. Therefore, the number of distinct processes sending a message of type "Check z" to a process P_i is bounded by d. By Lemma 1, every process completes all actions in one cycle within k time units, for some constant k. Therefore, P_i would be able to receive messages from all processes within kd time units after (R4) has partially stabilized. If $z_c[i] = 0$, then no process would send a message of type "Check z" to P_i as the code has stabilized and no node P_j has $code[j] = i$. So, $z[i]$ would be set correctly to 0. If $z_c[i] \neq 0$, then by the algorithm, z is set to the highest node to send the "Check z" message to P_i. This is indeed the correct value of $z[i]$ and this value would not change in future. Hence, *SSR* stabilizes z and (R4) in $O(d)$ time units.

■

Lemma 4 *The algorithm SSR stabilizes the variable f and also the constraints (R5) and (R3)(i) within $O(d)$ time units.*

Proof: Consider the execution of the algorithm once (R4) has stabilized. If constraint (R3)(i) is violated for some process, it would be rectified through the local check given for it in the algorithm. But this may not necessarily ensure that the value of f would be stabilized as f can be changed by the check on (R5). However, the check for (R5) sets the value of $f[i]$ to $z[i]+1$ if $z[i] > 0$. Since we know that $z[i]$ has stabilized and will not change, so the check for (R5) would always set it to the same value. From the definition of z and the fact that $code[n-1] = n$, it follows that $\forall i \neq n : z[i] \leq n-2$. This in turn implies that by changing f to follow the constraint (R5)(ii), we will not violate the constraint (R3)(i). So for both the cases, when $z[i] = 0$ and $z[i] \neq 0$, after one cycle f stabilizes to a value which satisfies the constraints (R5) and (R3)(i). Since one cycle requires $O(1)$ time units, the constraints (R5) and (R3)(i) would also stabilize within $O(d)$ time.

■

Lemma 5 *The algorithm SSR stabilizes (R1) within $O(d)$ time units.*

Proof: Consider the execution of the algorithm once f and *code* have stabilized. Then from the algorithm it is clear that the check for (R1) would stabilize through the check for (R1) in one cycle. Since, one cycle requires $O(1)$ time units, the constraint (R1) would stabilize within $O(d)$ time.

■

Theorem 4 *The algorithm SSR converges to a spanning tree in a finite number of moves irrespective of the initial state of the system and the order in which the processes execute the steps.*

Proof: For proving the theorem, it is sufficient to show that the data structures in the algorithm would satisfy the constraint set $\mathcal{R}$ after a finite number of steps and this follows from the previous lemmas.

■

Theorem 5 *The algorithm SSR stabilizes in $O(d)$ time, where d is the upper bound on the number of times a node appears in the code.*

Proof: By the previous lemmas, all the constraints in constraint set $\mathcal{R}$ are satisfied within $O(d)$ time units. Hence the algorithm *SSR* stabilizes in $O(d)$ time units.

■

We further note that the value of d would not be very large for a randomly chosen code. The problem of choosing the first $n-2$ numbers of *code* at random can be considered as the problem of randomly assigning $n-2$ balls to n bins. The following theorem is a standard result in probability theory [MR95][Theorem 3.1]:

Theorem 6 *If n balls are thrown randomly in n bins, then with the probability at least $1-\frac{1}{n}$, no bin has more than $\frac{e \log n}{\log \log n}$ balls.*

For a randomly chosen code, this theorem provides an upper bound for d and hence an upper bound on the stabilization time with high probability. This results in a very good stabilization time for our algorithm.

5.5.3 Maintaining $\mathcal{C}$

Maintaining $\mathcal{C}$ requires ensuring $R3(ii)$ in addition to $\mathcal{R}$. We introduce the variable g and enforce the constraints (D1)-(D3) listed in section 5.2. As discussed in the Section 5.2, one error in the data structures f and g can be corrected using the algorithm given in Figure 5. Including this module in the algorithm *SSR* gives us an $O(1)$ algorithm (Figure 8) which is capable of correcting many errors in the data structures. Unfortunately, this algorithm is not able to handle more than one correlated errors in f and g. When an error cannot be corrected by the $O(1)$ correction algorithm, the second check for the consistency of f and g fails for some node. This node sends out a message to every other node informing them to start the main correction algorithm.

Upon starting the main correction algorithm, every node sends out its z value to node P_n. Node P_n collects responses from every node and then establishes a mapping between the nodes which have $z = 0$ and the f values that have not been allocated. By allocated f values, we mean the f values which can be obtained as $z+1$, for some $z \neq 0$. The nodes which have $z \neq 0$ are assigned $f = z+1$. These results are communicated back to the nodes. In this case, the node P_n would have to do $O(n)$ work. After finishing the $O(n)$ correction algorithm, the nodes switch back to the normal correction algorithm. We will refer to this complete algorithm as *SSC*.

Clearly, there is a trade-off involved in choosing between the two algorithms. The algorithm for maintaining $\mathcal{R}$ is more efficient but gives weaker guarantees over the resulting spanning tree than the algorithm for maintaining $\mathcal{C}$.

5.6 Changing the Root Node

The algorithms *SSR* and *SSC* can be easily modified to allow the root node to change dynamically i.e. any node (not necessarily n) can become the root of the tree and the root can be changed during the operation of the algorithm. This can be achieved by changing the constraints (R2) and (R3)(i) in the following way:

(R2) $(\forall i : 1 \le i \le n-1 \Rightarrow 1 \le code[i] \le n)$
$\wedge (code[n] = 0)$

(R3)(i) $\forall i : i \ne code[n-1] \Rightarrow 1 \le f[i] < n$

The modified constraints are also easy to check and maintain. The algorithms which allow dynamic root can then be obtained by changing the algorithms *SSR* and *SSC* to accommodate checking for these new constraints instead of the old ones. In the next section we present an application which utilizes this feature.

6 Applications

The algorithm for maintaining constraint set $\mathcal{R}$ ensures that if the code is changed, then the spanning tree would stabilize to reflect that change. This property of the algorithm could be used by an application to purposefully change the spanning tree from time to time. As discussed earlier, every code of length $n-1$ represents a unique tree. If we were maintaining a tree isomorphic to the code tree (by maintaining the set of constraints $\mathcal{C}$), then a node wishing to change the tree could have changed its local code value. It can be proved that this would have resulted in the spanning tree being changed. But if we are just maintaining the set of constraints $\mathcal{R}$, then a code can represent multiple trees and similarly a single tree can be represented by a multiple codes. So changing the code value at a node may not always result in a change in the tree. To get around this problem, whenever a node i wishes to change the tree, it would change the value of $code[f[i]]$. This changes $parent[i] = code[f[i]]$ and hence

```
Pi::
 var
     code, parent, f, z: integer;

 Periodically do
     // Check (R2)
     if ((i = n−1) ∧ (code ≠ n))
         code = n
     if ((i = n) ∧ (code ≠ 0))
         code = 0
     if ((i ≠ n) ∧ ((code ≤ 0) ∨ (code > n)))
         code = random number between 1 and n

     // Check (R3)(i)
     if ((i ≠ n) ∧ ((f ≤ 0) ∨ (f ≥ n)))
         f = random number between 1 and n−1
     // First check for (R4)
     if (z ≠ 0)
         get code from node Pz
         if Pz.code ≠ i
             z = 0
     if (code ≠ 0)
         send ("Check z", i) to node code

     // Check (R5)
     if ((z ≠ 0) ∧ (f ≠ z+1))
         f = z+1
     if ((z = 0) ∧ (f ≤ z))
         f = random number between 1 and n−1

     // Check (R1)
     get code from node Pf
     if (Pf.code ≠ parent)
         parent = Pf.code

     // Second check for (R4)
     Upon receiving ("Check z", j)
         if (z < j)
             z = j
```

Figure 7: Algorithm *SSR* for maintaining the constraint set $\mathcal{R}$

P_i::
var
$code, parent, f, z$: integer;

Periodically do
// Check (R2), (R3)(i), (R4) - Same as in Figure 7

// Check (R5)
if $((z \neq 0) \wedge (f \neq z+1))$
$f = z+1$

// Check (R3)(ii)
get $g_1 = P_f.g$ from P_f and $f_1 = P_g.f$ from P_g
get f from P_{g_1} and g from P_{f_1}
if$((g_1 \neq i) \wedge (P_{g_1}.f \neq f))$
send ("Update g",i) to node P_f
if$((f_1 \neq i) \wedge (P_{f_1}.g \neq g))$
send ("Update f",i) to node P_g

// Second Check f and g
get $g_1 = P_f.g$ from P_f and $f_1 = P_g.f$ from P_g
if $((g_1 \neq i) \vee (f_1 \neq i))$
send ("Start Main") to all nodes

// Check (R1) - Same as in Figure 7

// Asynchronous message handling
Upon receiving ("Check z",j)
if $(z < j)$
$z = j$
Upon receiving ("Update g",j)
$g = j$
Upon receiving ("Update f",j)
$f = j$
Upon receiving ("Start Main")
start main correction algorithm

Figure 8: Algorithm *SSC* for maintaining the constraint set $\mathcal{C}$

the spanning tree changes. Note that this change may result in some more changes in the spanning tree as the parent of some other nodes may also get modified. Since the algorithm for maintaining the set $\mathcal{R}$ of constraints is efficient, this results in an efficient way of changing the tree. There could be many reasons for changing the spanning tree and we present three applications which require such a property.

- **Security**: Consider a scenario in which a set of nodes are contacting each other by using a tree for routing messages. For the system's security, this tree must not be revealed to the adversary. In case a security breach is suspected or after a regular interval of time, the tree must be changed and any node should be able to initiate this change without requiring active participation from other nodes. Our algorithm provides one such way. When a node i wishes to change the structure of the tree, it could just change the value of $code[f[i]]$ and initiate the correction algorithm. The ability to change the root node is critical here;otherwise, the adversary could always attack the fixed root node.

- **Reliability**: Consider a scenario in which we wish to relay messages through a spanning tree. Among the set of participating nodes, some nodes may be unreliable. It is advantageous to have the unreliable nodes in the leaf position as it would ensure that upon the failure of an unreliable node, the communication among other nodes is not disrupted. Such a constraint could be easily maintained by incorporating an additional check in algorithm in Figure 7. This check ensures that *code* takes values from the set of reliable nodes. Since this check would be a local check, the running time of the algorithm and the messages exchanged during the algorithm remains same.

- **Load Balancing**: Consider a scenario where a set of nodes are communicating through a spanning tree for an application like convergecast. In this case, a node has to do work proportional to size of its subtree which consumes resources like power, CPU etc. Since we are dealing with a completely connected topology, all the nodes are equally well connected and it is possible for a node to take up the job of another node. When a node wishes to reduce its load, it could its ask one of its child c to change the value of $code[f[c]]$ and hence change its load.

7 Core and Non-core data structures for other applications

In this section, we briefly mention two simple applications for utilizing the idea of *core* and *non-core* data structures.

7.1 Maintaining Permutation

Given a set of n nodes, we want to maintain the permutation of numbers from 1 to n. We discussed a way for checking the correctness of a permutation by using the functions f and g in section 5.2. Here we use the concept of *inversion vector* along with the above functions to maintain a permutation. The inversion vector of a permutation $(a_1, a_2, \dots, a_n)$ can be defined as $(t_1, t_2, \dots, t_n)$ where $t_k = |j|a_j > a_k, j < k|$. The important property of an inversion vector is that any sequence of n numbers $t_1, t_2, \dots, t_n$ satisfying the property $\forall i : t_i \leq i$ corresponds to a valid permutation. This opens the avenue for using the inversion vector as the *core* data structure for maintaining a permutation. A very simple way of using inversion vector is to combine it with f and g. A coordinator node keeps an inversion vector corresponding to the current permutation. The permutation itself is distributed across n nodes with each node maintaining the variables f and g. The permutation is checked for errors through consistency checks on f and g and in case a fault is detected, the coordinator regenerates the permutation corresponding to its current inversion vector. Note that the inversion vector of the coordinator might have also changed due to data faults. This solution is not very efficient and it leaves open a research problem of finding a true distributed solution of the problem.

8 Discussion

8.1 General Graphs

So far we had assumed the underlying graph to be completely connected. Let us now consider general graphs. Due to the nature of modern computer networks, the major overhead involved in communication using message passing is incurred at the OS level. So even if a process sends a message to another process that is more than one hop away, the message overhead can be assumed to be incurred completely at the sender and receiver. In this way any network topology with routing can be considered as a complete graph. Our algorithm can also be modified for applications which require the parent of a node to be its 1-hop neighbor. We just add a new constraint which requires a node to check if the parent assigned to it is a 1-hop neighbor. By adding this, the detection still remains $O(1)$ but correction becomes inefficient.

8.2 Exact Neville's Code Tree

In the theorem 3, we proved that by maintaining the constraint set $\mathcal{E}$, we can maintain the exact Neville's code tree. Constraint (R6) can be checked by maintaining a circular doubly linked list with the nodes having $z[i] = 0$ as its elements. Let the set of all such elements ordered according to the process numbers be denoted by L. Each element of the link list would have a *next* and *prev* pointer pointing to the next and the previous element, respectively, in the list. The next element in the list would be the node which occurs next to the node i in L. Similarly, the previous node would be the one to occur before node i in L. The only exceptions to the above rules are the *prev* pointer of first node in the list and the *next* pointer of the last node. To prevent a fragmented list to go undetected, we use the process P_1 as the sentinel node. So the *prev* pointer of the first node in the list and the *next* pointer of last node points to P_1. Similarly, P_1 also maintains its *prev* and *next* pointers. Now for checking (R6), we can just check for the consistency of the *prev* and *next* pointers along with (R6) for just the neighbors. Note that in this way we can detect faults in a link list in general which can be used for other applications as well.

8.3 Maintain a Trivial Spanning Tree

It is very reasonable for a reader to be thinking that if all the spanning trees constitute legal state, then why not maintain a trivial spanning tree like one in which all nodes $P_1 \dots P_{n-1}$ have P_n as parent ? The answer to this question again lies in the kind of application we are looking at. The applications considered in the section 6 required an application to keep changing its spanning tree and in that case such an algorithm would be suitable.

9 Conclusion and Future Work

In this paper we presented a new technique for maintaining spanning trees using labeled tree encoding. Our method requires $O(1)$ messages per node on average and also provides a method for changing the root of the tree dynamically. We also provide examples of using the self-stabilizing algorithm for some applications not related to fault tolerance. This work also demonstrates the use of the concept of *core* and *non-core* states for designing self-stabilizing algorithms. It would be interesting to extend this work for general topology. Another research direction could be to develop similar algorithm without requiring nodes to be labeled from 1 to n.

References

[AG94] A. Arora and M. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

[AK93] S. Aggarwal and S. Kutten. Time optimal self-stabilizing spanning tree algorithm. In *FSTTCS93 Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag LNCS:761*, pages 400–410, 1993.

[AKY91] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general networks. In *Proceedings of the 4th International Workshop on Distributed algorithms*, pages 15–28. Springer-Verlag New York, Inc., 1991.

[AS97] G. Antonoiu and P.K. Srimani. Distributed self-stabilizing algorithm for minimum spanning tree construction. In *European Conference on Parallel Processing*, pages 480–487, 1997.

[CD94] Z. Collin and S. Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297–301, 1994.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[DIM89] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems. In *MCC Workshop on Self-Stabilizing Systems*, 1989.

[DIM90] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *Proceedings of the ninth annual ACM symposium on Principles of Distributed Computing*, pages 103–117. ACM Press, 1990.

[DIM91] S. Dolev, A. Israeli, and S. Moran. Uniform self-stabilizing leader election. In *Proceedings of the 5th Workshop on Distributed Algorithms*, pages 167–180, 1991.

[DM01] N. Deo and P. Micikevicius. Prufer-like codes for labeled trees. *Congressus Numerantium*, 151:65–73, 2001.

[HC92] S. Huang and N. Chen. A self stabilizing algorithm for constructing breadth first trees. *Information Processing Letters*, 41:109–117, 1992.

[Joh97] C. Johnen. Memory efficient, self-stabilizing algorithm to construct bfs spanning trees. In *Proceedings of the sixteenth annual ACM symposium on Principles of Distributed Computing*, page 288. ACM Press, 1997.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[Nev53] E. H. Neville. The codifying of tree-structure. *Proceedings of Cambridge Philosophical Society*, 49:381–385, 1953.